The Report Committee for Abraham H. Bushmais
Certifies that this is the approved version of the following report:

Graph Based Unit Testing

APPROVED BY
SUPERVISING COMMITTEE:

Supervisors: _____
Khurshid, Sarfraz

_____
Krasner, Herb

Graph Based Unit Testing

By

Abraham H. Bushmais, B.S.

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

August, 2011

Dedication

I would like to dedicate this report to my wife and children. I want thank them for their patience, support and sacrifice during the last two years.

Acknowledgements

I would like to thank both Dr. Khurshid and Dr. Krasner for their help and guidance in producing this report.

Abstract

Graph Based Unit Testing

Abraham H. Bushmais, MSE

The University of Texas at Austin, 2011

Supervisor: Khurshid, Sarfraz

Automating test design can increase test suite accuracy and produce more reliable software. In this report we present a prototype tool that can aid developers in unit testing Java code. It automates test path construction based on two existing graph-based criteria. It uses basis path coverage and prime path coverage to produce test paths. Our main contribution in this report is to design and implement a tool that goes beyond the commonly used coverage tools today. Common graph-based coverage tools support statement coverage and sometimes branch coverage. Our tool support prime path coverage which subsumes a number of other graph-based coverage criteria, including statement and branch coverage. Our tool is a Java based Eclipse plug-in that operates at the class level. It processes each method in a given class to produce a CFG, cyclomatic complexity, a set of basis paths, a set of prime paths, and a set of test paths based on prime path coverage.

Table of Contents

List of Figures

List of Illustrations

Introduction

Continued growth in software infrastructure, where people come to rely on software in almost every interaction with their environment places heavy emphasis on software correctness. Transportation, energy distribution, communications, banking and health care all use software with increasing dependency. Software testing is the most commonly used methodology for validating quality of software and achieving an acceptable level of software correctness.

Once developers generate code to satisfy specifications and design constraints, the new code must be unit tested. Failures are likely more easily found and cheaper to fix at the implementation phase than later on in the development cycle. A software tester must come up with a series of test cases that will uncover as many faults as possible. Most, if not all, software failures must be corrected before delivering the new software to customers. Software testing represents the ultimate review of specification, design, and code [1]. However, locating faults in a software program is not always easy. The test engineer's challenge is to design test cases that will have a high probability of finding failures.

Software testing is expensive and labor intensive. The most time consuming activity of software testing is test design and construction. Software testing often requires more than 50% of software development costs [2]. One of the main goals of software testing is to automate as much as possible to reduce costs and increase reliability [2]. Automating test design increases reliability not just for the software being tested but also for the test suite itself. It reduces the chance of introducing human errors. Automation will also reduce the amount of effort and time needed to regenerate test suites due to system maintenance changes.

While software testing is extremely important, it should be noted that testing does not guarantee error free software. Testing can only show the presence and not the absence of

failure [2]. The problem of finding all failures in a program is undecidable. Software code is written by humans and humans make mistakes. For complex logic, there is no foolproof way to catch all mistakes made by humans. However, for most applications, that are not mission-critical, users of software will tolerate a certain amount of failure.

Software testing techniques in this report are presented in the context of unit testing source code. It is a type of white-box testing. Unit testing can uncover subtle errors that would be more expensive to uncover and fix later on in the development life cycle. Unit testing refers to assessing the software with respect to implementation [2]. It is testing the low level components, e.g. methods, in isolation. In this report, a unit refers to one method in a Java class. White-box testing refers to examining the code and deciding on logical paths to test. It is a static structural analysis. White-box testing is usually used for complex logic code that requires high levels of correctness.

White-box testing provides a good starting point. It is not a complete testing solution. White-box testing is complemented by black-box testing or functionality testing. Functionality testing crosses the boundaries of methods or even classes. Then there is integration testing and user acceptance testing. Each test category focuses on a different aspect of a software system. Each category can have its own techniques for better testing coverage.

Test cases derived from white-box testing are meant to ensure that every statement in a method is executed at least once and that every logical condition has been exercised. While white-box testing implies that exhaustive testing will yield error-free methods, exhaustive testing, in most scenarios, is not possible. Even for small methods, the number of possible logical paths to test can be very large. However, using specific coverage criteria, a test engineer can select a small but important set of test cases to get the desired level of software correctness.

Coverage Criteria

If it is hard or even impossible to eliminate all faults in software, then when can we consider testing to be complete?

Test engineers need systematic guidance in test design and in test construction. Formal coverage criteria provide this guidance. A formal coverage criterion provides for ways to identify what tests to execute, and what test inputs to use during the tests [2]. A formal coverage criterion likely enhances our chance of finding failures. It provides us with acceptable conditions to stop testing. It provides informal assurance that our tested software is of a certain quality and reliability.

Formal coverage criteria are a recipe for generating test coverage requirements [2]. For example, if our test coverage requirement is to execute every statement in a method, then through a test run, we can infer how much of the requirement did the test actually cover. For a given test coverage requirement, testing is adequate if it delivers a high enough coverage. Ideally, we want 100% coverage. The level of test adequacy is determined by the nature of the application under test, the business need, and available resources. It is usually a tradeoff between budget, time, and quality.

"Software testing begins by creating a graph of important objects and their relationships and then devising a series of tests that will cover the graph so that every object and relationship is exercised and errors are uncovered". [1] There are various formal coverage criteria. This report focus on graph-based criteria where we generate test requirements based on properties of a unit's graph. This graph represents flow of control as unit elements are traversed during execution. For example, node coverage depends on a graph property of nodes (statements). Using the node criterion we would want to cover as many nodes as possible. Different graph properties produce different coverage criteria. We will cover more on control flow graphs in the next section.

Not all coverage criteria are equal. Graph-based criteria are related to each other through subsumption [2], and some are stronger than others. Weak criterion tends to be quicker to test, but can miss more faults. On the other hand, strong criterion increases the cost of testing, but can find more faults. It is important to find an acceptable level of test adequacy.

A coverage criterion leads to test requirements. Test requirements resulting from formal coverage criteria should deliver "good" tests. A "good" test according to [3] has the following characteristics:

1. A good test has a high probability of finding errors.
2. A good test is not redundant.
3. A good test is best-of-breed.
4. A good test is neither too simple nor too complex.

CONTROL FLOW GRAPH (CFG)

A CFG [4] is commonly used during compilation, but we will use it in this report for static analysis. CFG forms the foundation of many coverage criteria [2].

A CFG is a graph abstraction of a method. It represents every statement and path that can be traversed during method execution. It is composed of nodes and edges connecting the nodes. Edges are directed and they represent control flow including branches and jumps. A node in the graph represents a basic block. A basic block is a sequential set of statements without any jumps. A Basic block can be one elementary statement or a single expression. Our tool constrained a node to represent one elementary statement or a single expression. The result is a statement-level CFG. Note that there can be many CFG abstractions of the same method.

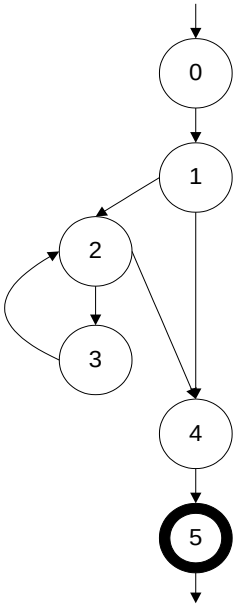
For a CFG to be useful in generating test paths, it must contain at least one initial node, one final node, and one intermediate node [2]. A test path must start at an initial node and end in some final node. A test path represents the execution of a test case. A test path

may cover several test cases. Note, that a test path with zero test cases is infeasible. An initial node corresponds to the starting block of the method. A final node is where control flow leaves the graph. A final node can be artificially added to the CFG to preserve the property that a final node post-dominates all other nodes. Although there can be many initial and final nodes in a CFG, our tool will only use single entry/single exit graphs.

For example, if we look at code for method `raise` x to the power k (figure to the right) and generate a control flow graph, we get the following CFG[1]:

```
public float raise(float x, int k) {
    float r = 1 ;              //0
    if (k >= 0) {              //1
        while (k-- != 0){      //2
            r = r * x ;        //3
        }
    }
    return r ;                 //4
}                              //5
```

raise () method code



raise () method CFG

A path is a sequence of nodes, where all the nodes in the sequence are connected. The length of a path is defined as the number of edges it contains [2]. A path of length 0 consists of one node. For example, from the `raise` CFG, [0, 1, 2, 4] is a valid path, while [0, 2, 4] is not (node 0 and 2 are not directly connected). A test path is a path that starts at the initial node and ends at the final node. For example, a valid test path would be [0, 1, 4, 5].

In graph coverage we track execution paths compared to certain paths we desire to cover. We define test requirements in terms of properties of test paths in a CFG. For example, visiting every edge in a CFG could be our graph coverage criterion - our test requirement. We then need a test set that satisfies our test requirement. After running our test cases we can determine how much of the requirement did the tests cover.

[1] The heavy bordered node is the final node.

Graph-based coverage criteria can be divided into two types [2]:

1. **Structural coverage criteria**

   Criteria of this type focus on structural properties of the CFG. Following Amman and Offutt [2] definitions, they include:

   a. **Node Coverage** (NC): our test requirement visits each reachable node (edge of length zero) form the initial node.
   b. **Edge Coverage** (EC): our test requirement visits each reachable edge of length one.
   c. **Edge-Pair Coverage** (EPC): our test requirement visits each reachable edge of length two. Edge coverage criteria can continue with the same idea to include edges of length three, four, and so on until we reach the maximum number of edges in a given CFG.
   d. **Prime Path Coverage** (PPC): our test requirement visits each prime path in the CFG. Prime paths will be covered in more detail in the next section.
   e. **Simple Round Trip Coverage** (SRTC): our test requirement contains at least one simple round trip path for each reachable node. A simple trip is a prime path of nonzero length that starts and ends at the same node.
   f. **Complete Round Trip Coverage** (CRTC): our test requirement contains all simple round trip paths for each reachable node.
   g. **Complete Path Coverage** (CPC): Our test requirement visits each reachable path in the CFG. This test would be useless if we have cycles in a CFG. Cycle produce infinite numbers of paths and hence an infinite number of requirements.

2. **Data flow coverage criteria**

   Criteria of this type are concerned with data definitions (defs) and data uses within a method. These criteria ensure that values created at some point in a method are actually used. Following Amman and Offutt [2] definitions, they include:

a. **All-Defs Coverage** (ADC): Our test requirement ensures that each def reaches at least one use.
b. **All-Uses Coverage** (AUC): Our test requirement ensures that each def reaches all uses through at least one path.
c. **All-du-Path Coverage** (ADUPC): Our test requirement ensures that each def reaches all uses through all paths. A du-path is a simple path that is also def-clear. A simple path is a path that has no internal loops, although the whole path can be one loop. A def-clear path is a du-pair path where the value of a def is not changed along the way to its use. A du-pair path is a path between a def of a value and its use.

PRIME PATH COVERAGE (PPC)

Complete path coverage (CPC) for methods with loops is infeasible, since we get an infinite number of paths to cover. The next best alternative to CPC, that can cover loops, is prime path coverage [2]. The idea is to use simple paths only. However, even small methods can have a fairly large number of simple paths and we need to minimize the number of paths to consider. Recall that a simple path is a path without internal loops, but the path itself may be a loop. A prime path is a maximal simple path that does not appear as a proper subpath of any other simple path. To avoid enumerating all simple paths for a given method, we focus on maximal length simple paths. Since any path can be created by composing a group of simple paths, taking only the maximal length simple paths will ensure maximum simple path coverage.

For example the `raise` method CFG (figure "CFG for raise () method"), will have the following prime paths:

[2, 3, 2] *[2], [3, 2, 3] *, [0, 1, 2, 3] ![3], [0, 1, 4, 5] !, [3, 2, 4, 5] !, [0, 1, 2, 4, 5] !

[2] * indicates a simple path that is a loop.
[3] ! indicates a simple path that ends at the final node.

Each of these prime paths represents a test requirement. Prime path [0, 1, 2, 4, 5] also represents a test path. The others can be extended to be test paths. Note that [2, 3, 2] and [3, 2, 3] are looping the same nodes. The reason we have two paths for the same loop is because PPC also satisfies Complete Round Trip Coverage (CRTC).

When creating test paths based on `raise` method's set of prime paths, one test path can cover many prime path test requirements, as long as we follow the same sequence of nodes. For example, test path [0, 1, 2, 3, 2, 4, 5] will cover four test requirements: [2, 3, 2], [3, 2, 3], [0, 1, 2, 3] and [3, 2, 4, 5]. Test path [0, 1, 2, 3, 2, 4, 5] can be covered by choosing a test case that uses parameters x to be any number and k = 1.

Prime path coverage criterion reduces the number of test paths, but it may produce infeasible prime paths. However, we can replace infeasible paths, with relevant feasible subpaths (look at best-effort touring in the Subsumption section).

BASIS PATH COVERAGE (BPC)

Depending on the number of branches in a method, the number of prime paths can be exponential. Setting up test cases to cover all prime paths may be resource intensive. An alternative is to use linearly independent paths, or basis paths. This method of testing was introduced by McCabe [5]. The idea is to find all linearly independent paths in a method and make those paths our test requirements. Test paths derived form basis path coverage criteria are guaranteed to execute each statement at least once and every condition at least twice (once on its true side and once on its false one). BPC is a weaker criterion than PPC, and choosing a weaker criterion produces the risk of overlooking some faults.

An independent path is any path through the method that introduces at least one new set of processing statements or conditions [1]. There are, potentially, many basis paths sets for a given method.

For example, using our tool on `method0` method (figure below), we get the following basis paths:

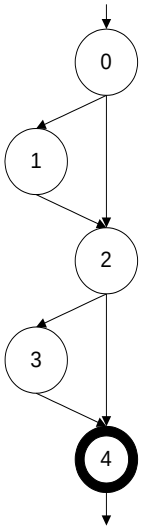
[0, 1, 2, 4] , [0, 2, 3, 4] , [0, 2, 4]

```
private void method0(int x) {
    if (x > 0)                                  //0
        System.out.println("Positive");         //1

    if (x < 0)                                  //2
        System.out.println("Negative");         //3
}                                               //4
```

method0() code

Note that all basis baths are test paths that start with the initial node and ends at the final node. The paths can be visually verified on the `method0()` CFG (figure on the right). If we take a look at the prime paths generated by the same tool for the same method:



method0 CFG

[0, 2, 4] !, [0, 1, 2, 4] !, [0, 2, 3, 4] !, [0, 1, 2, 3, 4] !

Path [0, 1, 2, 3, 4] is not considered in basis coverage. This path is not independent because it does not introduce any new nodes. It is simply a combination of already specified paths and traverses no new nodes. However, [0, 1, 2, 3, 4] is a prime path and it will be considered as a test requirement in prime path coverage.

Note there could be many sets of basis paths. This fact provides testers with the freedom of selecting convenient test paths, but how many do we need? What is our minimum? The answer is provided through cyclomatic complexity (see next section).

It may be straight forward to identify linearly independent paths of simple methods. However, for complex logic it is not so easy to determine basis paths. For more complex CFG, we would need an automated tool to get basis paths.

Cyclomatic Complexity (CC)

Cyclomatic Complexity is a software metric that provides a quantitative measure of the logical complexity of a program [1]. This metric was introduced by Thomas J. McCabe, Sr. [5]. Studies have shown that there are several implications to this metric [6]:

1. This quantity provides an upper bound for the number of tests that must be conducted to have edge coverage (EC).
2. This quantity provides a lower bound on the number of independent paths through a CFG (i.e. this is the number of basis paths). For example if CC = 3 then we need a set of 3 linearly independent paths to satisfy basis coverage.
3. A High CC measure implies low method cohesion. Cohesion measures the relatedness of functionality within a method. High cohesion produces more reliable and reusable methods.
4. The number of software defects is also correlated to CC measure. The higher the CC number, the more defects software tends to have.
5. Developers need to easily understand source code so that maintenance has less of a chance to introduce new errors. The higher the CC measure the more difficult it is to understand source code.

McCabe recommends that modules' complexity not exceed 10. If a module's CC exceeds 10, it needs to be split into smaller modules [5].

CC can be easily obtained by counting decision points in a method and adding one. For example, the `raise` method (code on page 5), and `method0` (code on page 9) would both have CC of 3. Another way to get CC is to calculate $e - n + 2$, where e is the number of edges in the CFG, and n is the number of its nodes. CC is always a positive number.

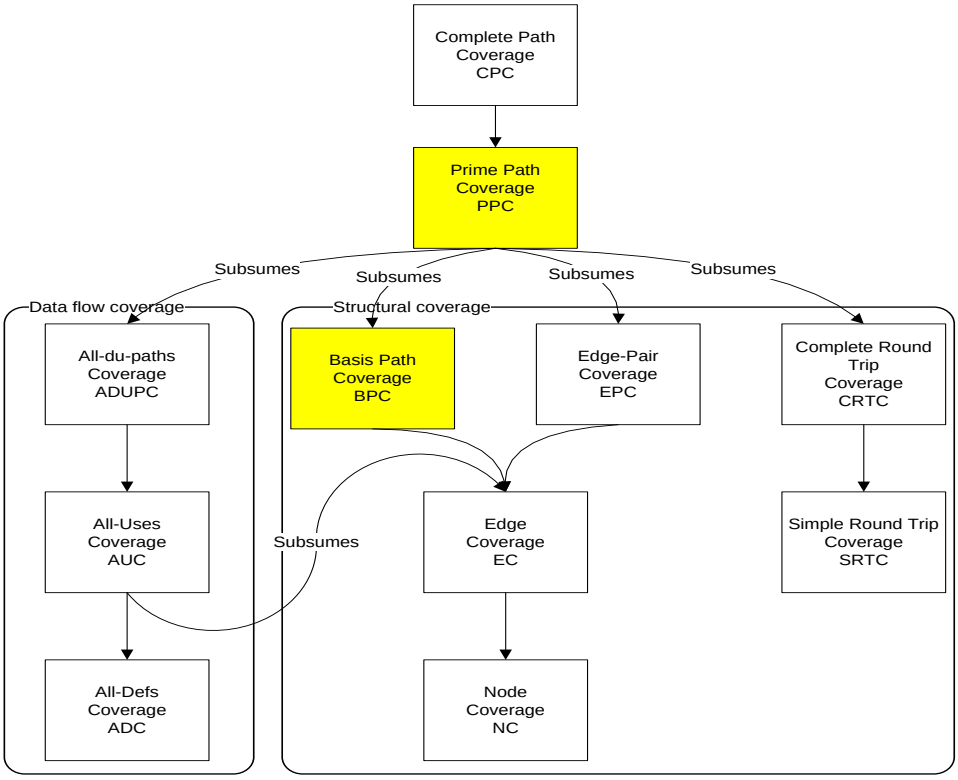
CC measurement can provide a red flag for methods that may require additional attention during testing. It can help focus the effort on more error-prone parts of a system.

SUBSUMPTION

Most coverage criteria relate to one another by subsumption [2]. A coverage criterion subsumes another, if every test requirement that satisfy the first criterion also satisfies the second. For example, complete edge coverage obviously implies node coverage, but not the other way around. In other words, satisfying edge/path coverage will guarantee that node/statement coverage is satisfied. This relationship is based on the assumption that all criteria coverage involved used best-effort touring.

Best-effort touring is the idea of meeting test requirements, first with strict tours (tours that adheres to the coverage criteria definition), and then allowing side-trips and detours for unmet test requirements. Side-trip is where we can leave a path temporarily from one node and return to the same path through the same node. Side-trips might be necessary if we get infeasible paths. Detours are where we can leave the path to visit other nodes, but we must come back to the same path and visit it in the same order of path nodes.

Amman and Offutt [2] contend that if every coverage criteria is satisfied using best-effort touring, then we would get the following subsumption relationship:



Subsumption relations among graph coverage criteria[4]

This relation was modified form [2] by adding basis path coverage. Note that prime path coverage subsumes all data flow and structural coverage criteria. It subsumes nine different coverage criteria in the figure above. If we compute prime path coverage, through subsumption, we are guaranteed coverage to all criteria subsumed by it. Computing prime path coverage is considerably simpler than analyzing data flow relationships [2].

Note subsumption relation between criteria does not imply similar relations between corresponding test paths. In other word, test paths that satisfy one criterion do not necessarily satisfy this criterion subsumed coverage criteria.

[4] This figure is adapted from [2], page 50.

Unit Testing Tool

Unit testing tool (UTT) will automate test path construction based on two graph-based criteria. It will use basis path coverage and prime path coverage to produce test paths. You might remember that basis path coverage subsumes edge coverage and node coverage, while prime path coverage subsumes all graph-based coverage presented in this report, with the exception of complete path coverage. Choosing which test paths to implement depends on the software being tested, availability of resources, and the business need.

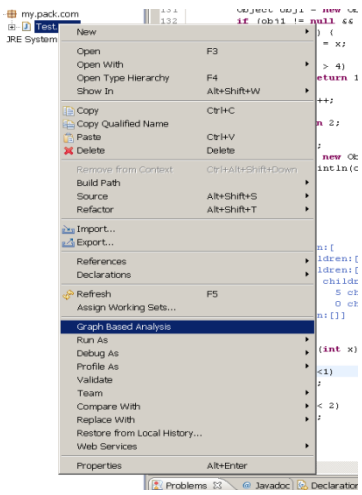
There are a variety of coverage tools in the industry. Yang et al. [7] compared 17 coverage based testing tools. The comparison primarily focused coverage measurement. The tools offered statement coverage and sometimes branch coverage. However, none of the tools, and as far as we know, no tool in the market, offers prime path coverage. Prime path coverage (PPC) subsumes a number of different coverage criteria that include data flow coverage and structural coverage. PPC is the strongest coverage criteria that can be practically applied in testing.

Commonly, coverage of 60% to 70% is considered acceptable because of the difficulty in increasing code coverage past 60% [7]. UTT provides assistance in achieving high code coverage in an effective way.

UTT is independent of requirement, or domain knowledge. All results are based on static analysis of Java source code. UTT also generates CC to aid developer in assessing a method's logic complexity as discussed earlier. The CC metric can help developer decide on simplifying their code before testing it.

This tool is for intra-methodic (or intra-procedural) testing. This means that testing is focused on one method (or one procedure) at a time. In other word, in the course of one test path in one method, testing will not jump to new method calls.

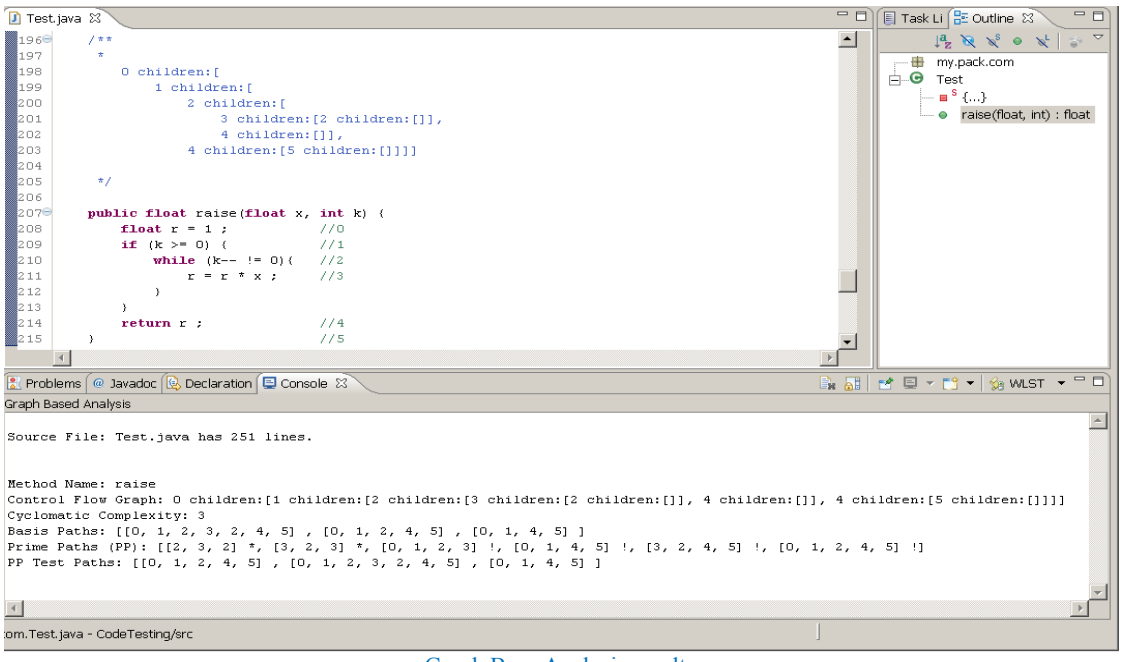
This prototype tool is written in Java and built as an Eclipse plug-in [8]. Eclipse is an extensible platform for building integrated development environments (IDEs). It provides a framework for controlling a set of tools working together to support programming tasks. Tool builders contribute to the Eclipse platform by adding Eclipse plug-ins, which are components that conform to Eclipse's plug-in contract.



Activating UTT

The tool is activated by right-clicking a java source file in the Eclipse IDE and selecting "Graph Based Analysis" (see figure above). After static code analysis, UTT will generate a control flow graph CFG for each method. Based on a CFG, UTT will also generate the method's cyclomatic complexity, a set of basis paths, the set of prime paths (PPC), and test paths satisfying PPC.

For example, if we select the Java file that contains our raise method in Eclipse and activate UTT, we get the following result:

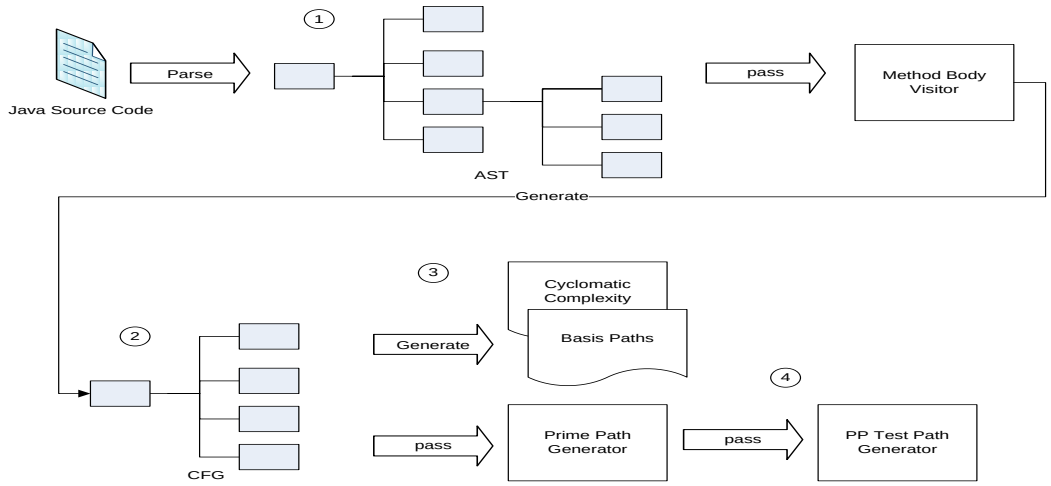


Graph Base Analysis results

Note that for this simple example, we used two different graph-based coverage criteria and we ended up with the same set of test paths.

UTT provides a set of test paths, but it does not provide data that can be used in the actual execution of the test cases.

UTT WORKFLOW



UTT Workflow

A typical workflow after UTT activation is as follows:

1. **Generating AST**: Source code is parsed into an abstract syntax tree. The AST contains a lot of information about each element of the source code.
2. **Generating CFG**: Using the Visitor pattern [9], the MethodBodyVisitor object will traverse and collect relevant node information to compose the method's control flow graph. Most of the AST element are ignored and are not necessary for UTT purposes.
3. **Generating paths**: The resulting CFG is then used to generate a value for cyclomatic complexity, and a set of basis paths. The CFG is passed to a PrimePathGenerator object that will generate a set of all prime paths.
4. **Generating test paths**: The resulting prime path set is passed to a TestPathGenerator object to generate a set of test paths.

Each of the steps above is elaborated on in the following sections.

GENERATING ABSTRACT SYNTAX TREE (AST)

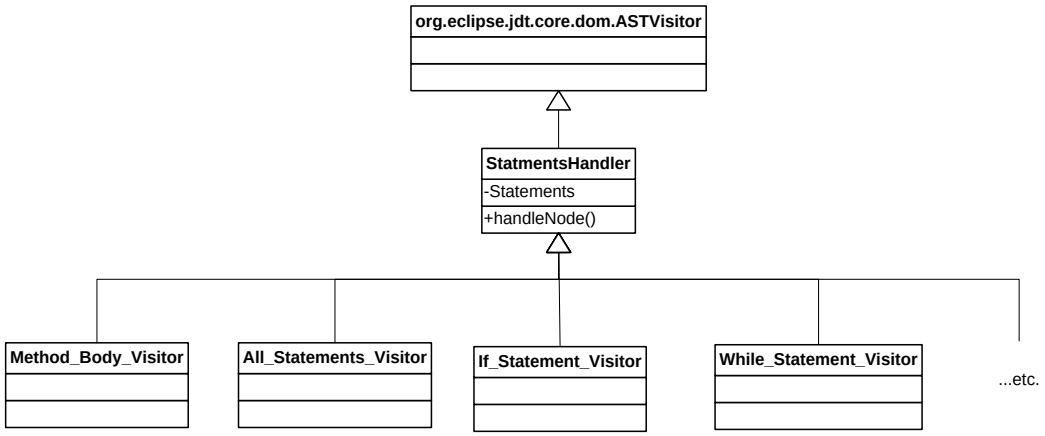
UTT builds a CFG for every method in a given class. It builds the CFG using Eclipse Abstract Syntax Tree (AST) [10]. AST is a data structure that provides semantic representation for a Java program, right down to every expression and statement in methods. AST provides the base framework for many powerful tools of the Eclipse IDE, including refactoring, Quick Fix and Quick Assist. An AST maps plain Java source code to a tree form that can be traversed for static analysis. An AST is more convenient than source code for automated analysis. Using AST in UTT provides a guarantee of well-structured code to analyze.

Using Eclipse Java core team packages, we parsed Java source code into an AST. We used `ASTParser` class form `org.eclipse.jdt.core.dom` package. The resulting AST data structure contains all elements in the method's body. Note this structure is detailed and verbose. Even small methods can result in complex ASTs.

GENERATING CONTROL FLOW GRAPH (CFG)

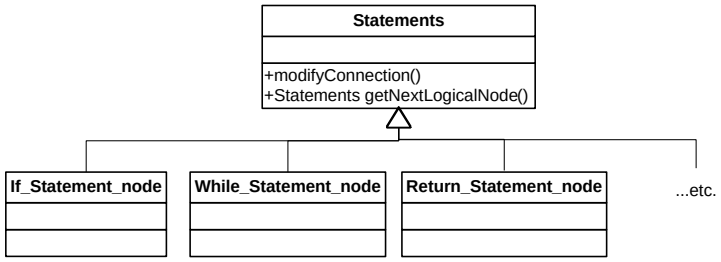
Once an AST for a given Java class is created, we divide the AST into blocks each block corresponds to a method's body. We loop through all blocks and using class `Method_Body_Visitor` we generate a CFG for each block. The `Method_Body_Visitor` class inherits form `ASTVisitor` class to use `preVisit(ASTNode node)` method to visit every AST element in the block. The `Method_Body_Visitor` class uses `StatmentsHandler's handleNode(ASTNode node)` method to call other visitors and combine the resulting partial CFGs into one complete CFG. The initial result is saved into a recursive data structure of type `Statements`. Each element that UUT can handle has its own visitor. The individual visitors return a partial CFG that represent the statements that were visited as nodes. Each node contains information about source code it represents, e.g. line number. Certain parts

of a Java statement are recursively visited while other parts are not. For example, in an if-statement we have an if-expression and a then-part. The else-part is optional. The body of a then-part or an else-part will always be recursively visited, but not the if-expression. This process will take care of nested if- statements. The same is true for other statements. A UML [11] diagram showing a partial list of UTT class inheritance is shown below:



UTT Visitor Inheritance

`Statements` is an abstract class that implements the design pattern Method Template [9]. When objects are instantiated they must be of a concrete type. Some concrete `Statements` types are shown in UTT Statements Inheritance figure. Every concrete class must implement the two methods `modifyConnection()` and `getNextLogicalNode()`. Method `modifyConnection()` will find the next logical node to connect to the current node. Both methods will have Java statement specific logic that knows what it means to get the next node in a while-statement as opposed to an if-statement. All this work takes place once method `connectSameLevelStatements()` from `Statements` is called. The final result is a control flow graph (CFG) representing one method's body.



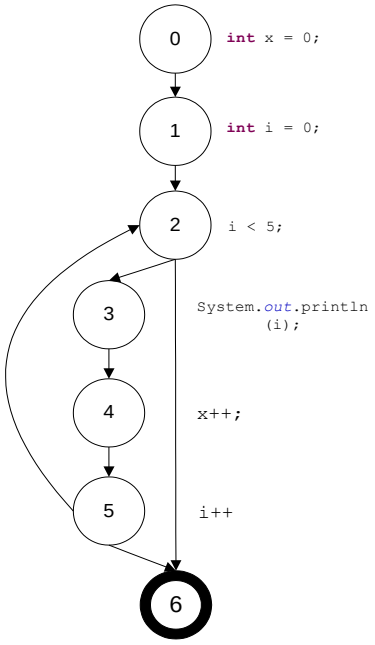
UTT Statements Inheritance

It worth noting, that the entire expression of a while-loop or an if-statement is represented in one node, while the For-loop signature takes up several nodes. For example `method21` is broken down into several nodes as shown in figure "method21 CFG". The figure shows the For-loop initializer, condition, and updater as separate nodes.

```
public void method21() {
    int x = 0;                          // 0
    for (int i = 0; i < 5; i++) {       // 1 2 5
        System.out.println(i);          // 3
        x++;                            // 4

    }
}                                       // 6
```

method21 Code



method21 CFG

The process of collecting nodes will ignore paths that represent jumps due to exceptions. It will also ignore exception statements. UTT currently does not handle the Java Do, Switch, or the enhanced For-loop statements. However UTT was written to follow the open/close principle [12] and it would be easy to add new Java statements for UTT to handle.

GENERATING CYCLOMATIC COMPLEXITY AND BASIS PATHS

Now that we have a statement-level CFG for a given method, we can calculate the method's Cyclomatic Complexity by using e − n + 2, where e is the number of edges in the CFG, and n is the number of its nodes.

To find the basis paths we use an adaptation of Poole algorithm [13]. The algorithm starts searching and recording all new nodes starting at the initial node of a CFG and recursively descends down all possible outgoing paths until it reaches the final node. Once it reaches the final node a basis path is outputted. The process terminates when there are no more paths to traverse.

GENERATING PRIME PATHS

Prime path discovery in UTT uses Amman and Offutt [2] algorithm described on pages 39 to 42. The algorithm starts by considering paths of length zero (nodes) to be prime path candidates, then paths of length one, two, and so on. It will progressively continue the process until we reach the maximum path length of a given CFG. The algorithm goes through a pruning process with the addition of each new prime path. If any existing prime path is contained in a new one, we remove it before adding the new one.

Generating test paths

Once we have a set of prime paths, we can generate the corresponding test paths. We followed Amman and Offutt [2] recommendation. We started with the longest prime path and extended it on both ends to include the initial and final nodes. The process continues with the remaining longest prime path, until we are done with the entire set. Prime path extension process depends on the CFG at hand. Using the `Statements` object UTT is able to extend a prime path to follow the CFG on both ends of the path.

Each new test path is checked against existing paths for redundancy. If the entire new path, as a sequence of nodes, already exists in our test path set, it is ignored.

Prime paths covering a loop will generate similar test paths. These prime paths are pruned out before the process of generating test paths starts.

For example, Java code for Euclid's algorithm for finding greatest common divisors is shown in figure on the right.

```
// Euclid's algorithm for finding greatest common divisors.
public int greatestCommonDivisor(int m, int n){
    int r = 1;              //0
    if (m > 0 && n > 0){    //1
        if (n > m){         //2
            r = m;          //3
            m = n;          //4
            n = r;          //5
        }
        r = m % n;          //6
        while(r != 0){      //7
            m = n;          //8
            n = r;          //9
            r = m % n;      //10
        }
    }
    return r;               //11
}                           //12
```

Euclid's algorithm

Running UTT on the class containing this method will yield the following information:

```
Method Name: greatestCommonDivisor
Control Flow Graph:  0 children:[1 children:[2 children:[3 children:[4 children:[5 children:[6 children:[]]]],
                                                        6 children:[7 children:[8 children:[9 children:[10 children:[7 children:[]]]],
                                              11 children:[]]]], 11 children:[12 children:[]]]
Cyclomatic Complexity: 4
Basis Paths: [[0, 1, 2, 3, 4, 5, 6, 7, 11, 12] , [0, 1, 2, 6, 7, 8, 9, 10, 7, 11, 12] , [0, 1, 2, 6, 7, 11, 12] , [0, 1, 11, 12] ]
Prime Paths (PP): [[0, 1, 11, 12] !, [7, 8, 9, 10, 7] *, [8, 9, 10, 7, 8] *, [9, 10, 7, 8, 9] *, [10, 7, 8, 9, 10] *, [8, 9, 10, 7, 11, 12] !,
                   [0, 1, 2, 6, 7, 11, 12] !, [0, 1, 2, 6, 7, 8, 9, 10] !, [0, 1, 2, 3, 4, 5, 6, 7, 11, 12] !, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] !]
PP Test Paths: [[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 7, 11, 12] , [0, 1, 2, 3, 4, 5, 6, 7, 11, 12] , [0, 1, 2, 6, 7, 8, 9, 10, 7, 11, 12] ,
                [0, 1, 2, 6, 7, 11, 12] , [0, 1, 11, 12] ]
```

The resulting control flow graph can be rewritten with more lines and indentation like this:

```
0 children:[
    1 children:[
        2 children:[
            3 children:[4 children:[5 children:[6 children:[]]]],
            6 children:[
                7 children:[
                    8 children:[9 children:[10 children:[7 children:[]]]],
                    11 children:[]]]],
        11 children:[12 children:[]]]
```

It can be read as follows: node 0 can only go to node 1. Node 1 can go to either node 2 or node 11. Node 2 can either go to node 3 or node 6, and so on.

A CFG representing `greatestCommonDivisor()` method is also shown in figure 1 in the appendix.

Note the resulting prime path set:

[0, 1, 11, 12] !, [7, 8, 9, 10, 7] *, [8, 9, 10, 7, 8] *, [9, 10, 7, 8, 9] *, [10, 7, 8, 9, 10] *, [8, 9, 10, 7, 11, 12] !, [0, 1, 2, 6, 7, 11, 12] !, [0, 1, 2, 6, 7, 8, 9, 10] !, [0, 1, 2, 3, 4, 5, 6, 7, 11, 12] !, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] !

All paths in red are derivatives of the same loop. UTT will select the first path [7, 8, 9, 10, 7] to represent the others. Following our algorithm for test path generation, we extend [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10] to get [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 7, 11, 12]. This test path will cover prime paths [8, 9, 10, 7, 11, 12] and [7, 8, 9, 10, 7] (which includes all derivatives). Note that one test path in this example has covered six prime paths.

IMPLEMENTING TEST PATHS

UTT will generate a set of test paths. Testers need to find input values that will cause execution flow to follow our test path. To test individual methods we may have to temporarily change private methods to public. Another way to test private methods is to incorporate a test driver within the class we are testing. We may also have to temporarily ignore pre/post conditions to make all test paths feasible.

OBSERVATIONS

In general, experienced Java developers follow well established rules and conventions about design and programming. These conventions include Object Oriented (OO) design patterns [9] and programming best practice, e.g. loose coupling and highly cohesive modules. The objective is to produce extensible code with fewer errors. As Beck [14] also suggested, to improve programs, you need to eliminate duplicate code, conditional logic and complex methods. This would suggest that experienced developers will produce code with low cyclomatic complexity.

There are several UTT run examples in the appendix. In addition, while evaluating and testing our tool, we ran UTT against a lot of real world methods. Our experience with UTT has led to the following observations:

- All methods that had CC of 2 (or less) shared the same test paths for both basis path coverage and prime path coverage.
- Most CC 3 methods shared the same test paths for both basis path coverage and prime path coverage. In fact, for methods that did not contain loops, CC 4 methods (and less) shared the same test paths for both basis path and prime path coverage.

While this is not a case study, we think it is safe to say that, in practice, for low cyclomatic complexity OO methods, the level of effort to implement prime path test cases compared to basis paths test cases will be about the same.

UTT EXTENSIONS

There are additional features that can be added to UTT to make more appealing. One feature we considered, and later dropped due to time constrains, is instrumentation [15].

For a coverage tool, UTT does not provide users with coverage information after test implementation. Code instrumentation allows us to get test coverage information after running the implemented test paths. One way to implement instrumentation is to use Java Management Extensions (JMX) [16] to trigger events on byte code execution within the Java virtual machine. A map between the byte code and the line numbers of the source code can be used to track which CFG nodes in a test path have actually been visited. This tracking can take place while executing JUnit [17] test cases. The JUnit framework will have to be extended to load the JMX agent before the start of a test run. JUnit test cases will still need to be setup by testers. This approach keeps UTT fixable, in terms of node selections for CFGs, and reduce the runtime overhead of code coverage.

Another interesting feature is test input generation. For methods that take integer (int), float, or real arguments we can use symbolic execution [18]. UTT can be extended by JPF [18] to generate test inputs that match our test paths only, and then execute the generated test cases. A UTT user will see the final run results and the failed test cases, if any.

Displaying coverage reports is an important feature of any coverage tool. UTT was implemented as an Eclipse plug-in because of the friendly graphic interface that Eclipse provides. A user interface can be a decisive element for a tool's usability [7]. Similarly a run results need to be shown in an easy to understand, friendly, way.

Conclusions

The most time consuming activity of software testing is test design and construction. To design test cases that will have high quality, test engineers often use formal criteria for guidance in test design and in test construction. Automating test design based on formal coverage criteria will likely not only increase reliability for the software being tested but also increase maintainability for the test suite itself. Automation will also reduce the amount of time needed for testing and therefore increase productivity.

Unit Testing tool (UTT) automates test design using formal criteria. UTT reduces the problem, and the effort, of designing and constructing test scenarios, to finding input values that satisfy test paths. These test paths are based on formal coverage criteria that likely provide high quality tests.

A key novelty of UTT is its support of prime path coverage (PPC). PPC subsumes a number of different coverage criteria that include data flow coverage and structural coverage. PPC is among the strongest coverage criteria that can be practically applied to software testing.

UTT's cyclomatic complexity (CC) measure provides developers with the opportunity to modify code and simplify it before proceeding with test implementation. Additionally, in practice, for low CC Java methods, implementing PPC test cases is no more expensive than implementing basis path coverage (BPC) test cases. This implies that, for code written to follow object oriented design principles, we can get much stronger coverage for about the same price of basis path coverage.

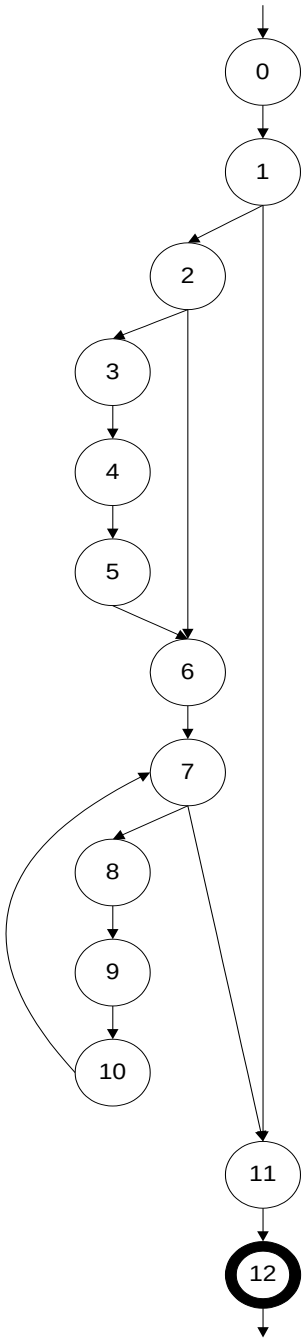
Appendix

GREATEST COMMON DIVISOR:

The statement-level control flow graph of `greatestCommonDivisor()` method:

```
// Euclid's algorithm for finding greatest common divisors.
public int greatestCommonDivisor(int m, int n){
    int r = 1;                  //0
    if (m > 0 && n > 0){        //1
        if (n > m){             //2
            r = m;              //3
            m = n;              //4
            n = r;              //5
        }
        r = m % n;              //6
        while(r != 0){          //7
            m = n;              //8
            n = r;              //9
            r = m % n;          //10
        }
    }
    return r;                   //11
}                               //12
```

Figure 1



UTT EXAMPLE RUNS: (PAGES 24 - 27)

Method13:

```
public void method13(int x, int g) {
    ;// ignored
    try {
        if (x > 0) {     // 0
            x--;         // 1
            g = x * x;   // 2
        } else {
            x++;         // 3
            g = x / x;   // 4
        }
    } catch (Exception e) {
        method14(x); // ignored
    }

}                        // 5
```

Cyclomatic Complexity: 2
Basis Paths: [[0, 1, 2, 5] , [0, 3, 4, 5]]
Prime Paths: [[0, 1, 2, 5] !, [0, 3, 4, 5] !]
PP Test Paths: [[0, 3, 4, 5] , [0, 1, 2, 5]]

Method3:

```
public void method3(int x, int g) {

    if (x > 0 && g > 0) {    //0
        if (g > 1) {         //1
            g = g + 1;       //2
            g++;             //3
            if (g > 10)      //4
                g++;         //5
            else
                g = g + g;   //6
        } else
            g--;             //7
    } else
        x = x + g;           //8

    System.out.println(x);   //9
}                            //10
```

Cyclomatic Complexity: 4
Basis Paths:[[0, 1, 2, 3, 4, 5, 9, 10] , [0, 1, 2, 3, 4, 6, 9, 10] , [0, 1, 7, 9, 10] , [0, 8, 9, 10]]
Prime Paths: [[0, 8, 9, 10] !, [0, 1, 7, 9, 10] !, [0, 1, 2, 3, 4, 5, 9, 10] !, [0, 1, 2, 3, 4, 6, 9, 10] !]
PP Test Paths: [[0, 1, 2, 3, 4, 6, 9, 10] , [0, 1, 2, 3, 4, 5, 9, 10] , [0, 1, 7, 9, 10] , [0, 8, 9, 10]]

Method10:

```
public int method10(int x, int y) {
    if (y > 0) {              //0
        x = x++;              //1
        if (x > 0) {          //2
            y++;              //3
            if (y > x) {      //4
                return x;     //5
            }
        }
    }
    return y;                 //6
}                             //7
```

Cyclomatic Complexity: 4
Basis Paths: [[0, 1, 2, 3, 4, 5, 7] , [0, 1, 2, 3, 4, 6, 7] , [0, 1, 2, 6, 7] , [0, 6, 7]]
Prime Paths: [[0, 6, 7] !, [0, 1, 2, 6, 7] !, [0, 1, 2, 3, 4, 5, 7] !, [0, 1, 2, 3, 4, 6, 7] !]
PP Test Paths: [[0, 1, 2, 3, 4, 6, 7] , [0, 1, 2, 3, 4, 5, 7] , [0, 1, 2, 6, 7] , [0, 6, 7]]

Method14:

```
public int method14(int x) {
    if (x > 0)            //0
    {
        x++;              //1
        return 1;         //2
    } else if (x > 1) {   //3
        return 2;         //4
    } else if (x > 2) {   //5
        return 3;         //6
    } else {
        return 4;         //7
    }
}                         //8
```

Cyclomatic Complexity: 4
Basis Paths: [[0, 1, 2, 8] , [0, 3, 4, 8] , [0, 5, 6, 8] , [0, 7, 8]]
Prime Paths: [[0, 7, 8] !, [0, 1, 2, 8] !, [0, 3, 4, 8] !, [0, 5, 6, 8] !]
PP Test Paths: [[0, 5, 6, 8] , [0, 3, 4, 8] , [0, 1, 2, 8] , [0, 7, 8]]

Method6:

```
public void method6(int x) {
    while (x > 0) {      // 0
        while (x < 10)   // 1
            x = x + 1;   // 2
        x = x + x;       // 3
        while (x > 10)   // 4
            x = x - 1;   // 5
        x--;             // 6
    }
}                        // 7
```

Cyclomatic Complexity: 4
Basis Paths: [[0, 1, 2, 1, 3, 4, 6, 0, 7] , [0, 1, 3, 4, 5, 4, 6, 0, 7] , [0, 1, 3, 4, 6, 0, 7] , [0, 7]]
Prime Paths: [[1, 2, 1] *, [2, 1, 2] *, [4, 5, 4] *, [5, 4, 5] *, [2, 1, 3, 4, 5] !, [5, 4, 6, 0, 7] !, [0, 1, 3, 4, 6, 0] *, [1, 3, 4, 6, 0, 1] *, [1, 3, 4, 6, 0, 7] !, [2, 1, 3, 4, 6, 0] !, [3, 4, 6, 0, 1, 2] !, [3, 4, 6, 0, 1, 3] *, [4, 6, 0, 1, 3, 4] *, [5, 4, 6, 0, 1, 2] !, [5, 4, 6, 0, 1, 3] !, [6, 0, 1, 3, 4, 5] !, [6, 0, 1, 3, 4, 6] *]
PP Test Paths: [[0, 1, 3, 4, 5, 4, 6, 0, 1, 3, 4, 6, 0, 7] , [0, 1, 3, 4, 5, 4, 6, 0, 1, 2, 1, 3, 4, 6, 0, 7] , [0, 1, 3, 4, 5, 4, 6, 0, 7] , [0, 1, 2, 1, 3, 4, 5, 4, 6, 0, 7]]

Method8:

```
public void method8() {
    int x = 0;                          //0
    for (int i = 0; i < 10; i++) {      //1 2 8
        x++;                            //3
        for (int j = 0; j < 10; j++) {  //4 5 7
            System.out.print(j);        //6
        }
    }
}                                       //9
```

Cyclomatic Complexity: 3
Basis Paths: [[0, 1, 2, 3, 4, 5, 6, 7, 5, 8, 2, 9] , [0, 1, 2, 3, 4, 5, 8, 2, 9] , [0, 1, 2, 9]]
Prime Paths: [[0, 1, 2, 9] !, [5, 6, 7, 5] *, [6, 7, 5, 6] *, [7, 5, 6, 7] *, [2, 3, 4, 5, 8, 2] *, [3, 4, 5, 8, 2, 3] *, [3, 4, 5, 8, 2, 9] !, [4, 5, 8, 2, 3, 4] *, [5, 8, 2, 3, 4, 5] *, [6, 7, 5, 8, 2, 9] !, [8, 2, 3, 4, 5, 8] *, [0, 1, 2, 3, 4, 5, 8] !, [6, 7, 5, 8, 2, 3, 4] !, [8, 2, 3, 4, 5, 6, 7] !, [0, 1, 2, 3, 4, 5, 6, 7] !]
PP Test Paths: [[0, 1, 2, 3, 4, 5, 6, 7, 5, 8, 2, 9] , [0, 1, 2, 3, 4, 5, 6, 7, 5, 8, 2, 3, 4, 5, 8, 2, 9] , [0, 1, 2, 3, 4, 5, 8, 2, 9] , [0, 1, 2, 9]]

Method19:

```
public void method19(int x) {
    if (x == 0) {            //0
        return;              //1
    }
    while (x < 10) {         //2
        x++;                 //3
        while (x < 7) {      //4
            if (x == 5) {    //6
                continue;    //7
            }
            x++;             //8
        }
        x = x + x;           //9
        while (x < 15) {     //10
            x = x - 1;       //11
            if (x == 5) {    //12
                break;       //13
            }
        }
    }
}                            //14
```

Cyclomatic Complexity: 7
Basis Paths: [[0, 1, 13] , [0, 2, 3, 4, 5, 6, 4, 8, 9, 2, 13] , [0, 2, 3, 4, 5, 7, 4, 8, 9, 2, 13] , [0, 2, 3, 4, 8, 9, 10, 11, 12, 2, 13] , [0, 2, 3, 4, 8, 9, 10, 11, 9, 2, 13] , [0, 2, 3, 4, 8, 9, 2, 13] , [0, 2, 13]]
Prime Paths: [[0, 1, 13] !, [0, 2, 13] !, [4, 5, 6, 4] *, [4, 5, 7, 4] *, [5, 6, 4, 5] *, [5, 7, 4, 5] *, [6, 4, 5, 6] *, [6, 4, 5, 7] !, [7, 4, 5, 6] !, [7, 4, 5, 7] *, [9, 10, 11, 9] *, [10, 11, 9, 10] *, [11, 9, 10, 11] *, [10, 11, 9, 2, 13] !, [0, 2, 3, 4, 5, 6] !, [0, 2, 3, 4, 5, 7] !, [0, 2, 3, 4, 8, 9] !, [2, 3, 4, 8, 9, 2] *, [3, 4, 8, 9, 2, 3] *, [3, 4, 8, 9, 2, 13] !, [4, 8, 9, 2, 3, 4] *, [8, 9, 2, 3, 4, 8] *, [9, 10, 11, 12, 2, 13] !, [9, 2, 3, 4, 8, 9] *, [2, 3, 4, 8, 9, 10, 11] !, [5, 6, 4, 8, 9, 10, 11] !, [5, 6, 4, 8, 9, 2, 3] !, [5, 6, 4, 8, 9, 2, 13] !, [5, 7, 4, 8, 9, 10, 11] !, [5, 7, 4, 8, 9, 2, 3] !, [5, 7, 4, 8, 9, 2, 13] !, [8, 9, 2, 3, 4, 5, 6] !, [8, 9, 2, 3, 4, 5, 7] !, [10, 11, 9, 2, 3, 4, 8] !, [10, 11, 9, 2, 3, 4, 5, 6] !, [10, 11, 9, 2, 3, 4, 5, 7] !, [9, 10, 11, 12, 2, 3, 4, 5, 6] !, [9, 10, 11, 12, 2, 3, 4, 5, 7] !, [9, 10, 11, 12, 2, 3, 4, 8, 9] *]
PP Test Paths: [[0, 2, 3, 4, 8, 9, 10, 11, 12, 2, 3, 4, 8, 9, 2, 13] , [0, 2, 3, 4, 8, 9, 10, 11, 12, 2, 3, 4, 5, 7, 4, 8, 9, 2, 13] , [0, 2, 3, 4, 8, 9, 10, 11, 12, 2, 3, 4, 5, 6, 4, 8, 9, 2, 13] , [0, 2, 3, 4, 8, 9, 10, 11, 9, 2, 3, 4, 5, 7, 4, 8, 9, 2, 13] , [0, 2, 3, 4, 8, 9, 10, 11, 9, 2, 3, 4, 5, 6, 4, 8, 9, 2, 13] , [0, 2, 3, 4, 5, 7, 4, 8, 9, 2, 3, 4, 8, 9, 2, 13] , [0, 2, 3, 4, 5, 7, 4, 8, 9, 10, 11, 9, 2, 13] , [0, 2, 3, 4, 5, 6, 4, 8, 9, 2, 3, 4, 8, 9, 2, 13] , [0, 2, 3, 4, 5, 6, 4, 8, 9, 10, 11, 9, 2, 13] , [0, 2, 3, 4, 8, 9, 10, 11, 12, 2, 13] , [0, 2, 3, 4, 5, 6, 4, 5, 7, 4, 8, 9, 2, 13] , [0, 2, 13] , [0, 1, 13]]

Bibliography

1. **Roger S. Pressman, Darrel Ince.** *Software Engineering: A Practitioner's Approach.* s.l. : McGraw-Hill, 2000.
2. **Paul Ammann, Jeff Offutt.** *Introduction to Software testing.* New York : Cambridge University Press, 2008.
3. **Cem Kaner, Jack Falk, Hung Quoc Nguyen.** *Testing Computer Software.* s.l. : John Wiley & son, 1999.
4. **Henry F. Ledgard, Michael Marcotty.** A genealogy of control structures. *Communications of the ACM.* 1975, Vol. 18, 11.
5. **McCABE, THOMAS J.** A Complexity Measure. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING.* 1976.
6. **Wikipedia.** Cyclomatic complexity. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/Cyclomatic_complexity.
7. *A Survey of Coverage Based Testing Tools.* **Qian Yang, J. Jenny Li, and David Weiss.** New York : ACM, 2006. ISBN:1-59593-408-1.
8. **Bolour, Azad.** Notes on the Eclipse Plug-in Architecture. [Online] Bolour Computing, 2003. http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html.
9. **Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.** *Design Patterns: Elements of Reusable Object-Oriented Software* . s.l. : Addison-Wesley, 1995.
10. **Eclipse.** FAQ What is an AST? *wiki.eclipse.org.* [Online] http://wiki.eclipse.org/FAQ_What_is_an_AST.
11. **Wikipedia.** Unified Modeling Language. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/Unified_Modeling_Language.
12. —. Open/closed principle. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/Open/closed_principle.
13. **Poole, Joseph.** A Method to Determine a Basis Set of Paths to Perform Program Testing. *U.S. Department of Commerce/National Institute of Standards and Technology.* [Online] November 1995. http://hissa.nist.gov/publications/nistir5737/index.html. NISTIR 5737.
14. **Beck, Kent.** *Smalltalk Best Practice Patterns Volume 1: Coding.* 1996.
15. **Wikipedia.** Instrumentation (computer programming). *Wikipedia.* [Online] http://en.wikipedia.org/wiki/Instrumentation_%28computer_programming%29.
16. **Oracle.** Oracle. *Java Management Extensions (JMX) Technology.* [Online] http://www.oracle.com/technetwork/java/javase/tech/javamanagement-140525.html.
17. **JUnit.** JUnit Resources for Test Driven Development. [Online] http://www.junit.org/.
18. **NASA.** Advanced Testing. *NASA.* [Online] http://ti.arc.nasa.gov/tech/rse/vandv/advtesting/.